C# 프로그래밍

- 3 주차 (2강)

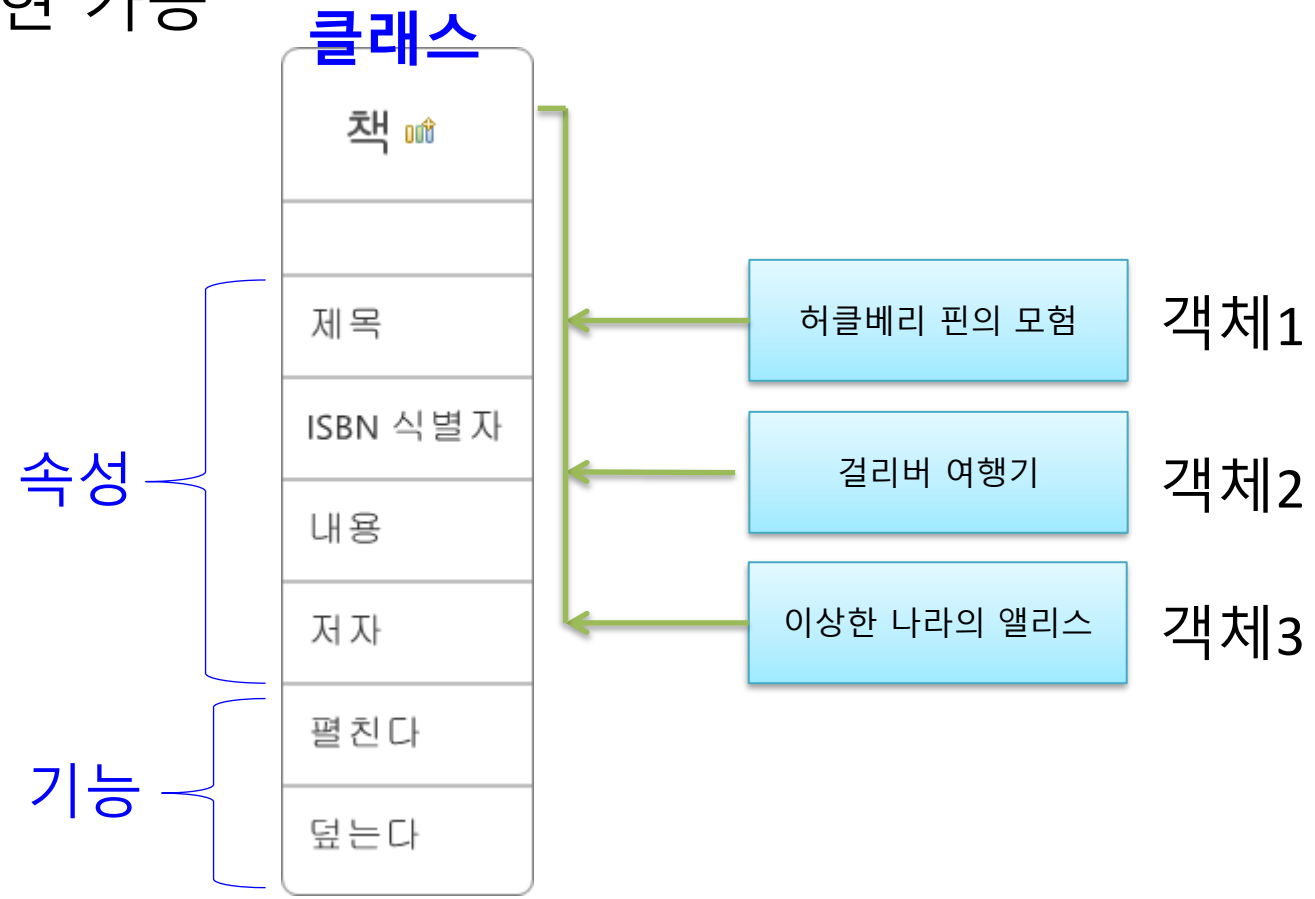
SEJONG UNIVERSITY

클래스 (class)

- 객체지향 프로그래밍
- 클래스 선언과 객체 생성
- 생성자와 종료자
- 메소드

객체지향(Object Oriented) 프로그래밍

- 코드 내의 모든 것을 객체로 표현하고자 하는 프로그래밍 패러다임
- 현실 세계의 모든 것들이 객체 (object)
- 각 객체의 특징은 속성과 기능으로 표현 가능
  - ✓ C#에서는 속성은 데이터로 기능은 메소드로 표현
- 클래스 (class)
  - ✓ 객체를 만들기 위한 '청사진' 또는 '틀'
  - ✓ 예를 들어, '책'이라는 개념은 '틀'
  - ✓ '걸리버 여행기'는 '책'이라는 '틀'이 실제화 된 객체



클래스 선언

- 클래스는 class 키워드를 이용해서 선언
  - ✓ 필드: 클래스 안에 선언된 변수
  - ✓ 멤버: 필드, 메서드, 프로퍼티, 이벤트 등 클래스 내에 선언되는 요소
- 예를 들어, 고양이를 추상화해서 클래스로 표현
  - ✓ 필드: 이름, 색깔
  - ✓ 기능: 야옹

사용 형식

```
class 클래스이름
{
    // 데이터와 메소드
}
```

사용 예제

```
class Cat
{
    public string Name;      // 필드1
    public string Color;     // 필드2

    public void Meow()       // 메소드
    {
        Console.WriteLine($"{Name} : 야옹");
    }
}
```

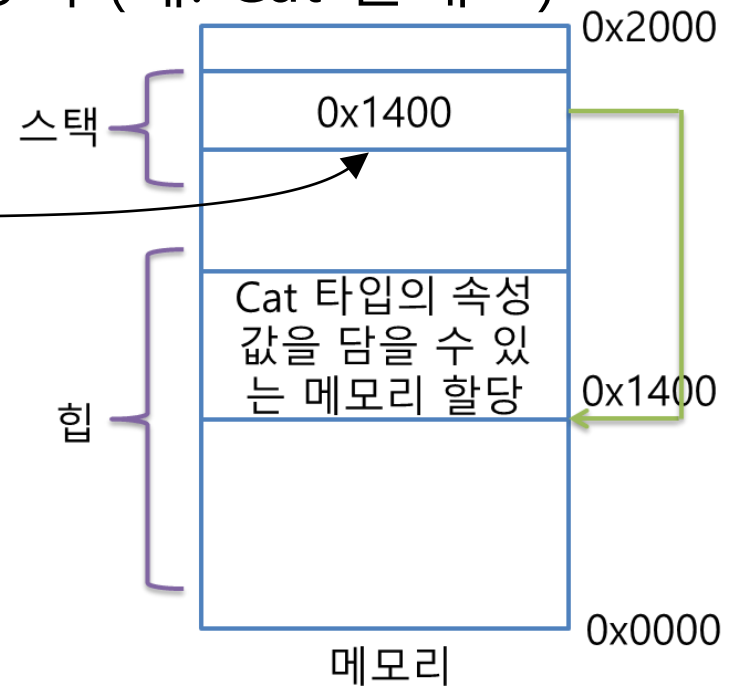
객체 생성

- 클래스는 복합 데이터 형식 (참조 형식)
  - ✓ 코드에서 보는 클래스는 또 하나의 데이터 형식
  - ✓ 예를 들어, string은 C#에서 이미 정의된 문자열을 다루는 클래스

```
string a = "123";   // a는 string의 객체 또는 인스턴스
string b = "Hello"; // b는 string의 객체 또는 인스턴스
```

- 개발자는 원하는 모든 객체의 타입을 새롭게 정의 (예: Cat 클래스)
  - ✓ Cat 의 객체 kitty 생성
  - ✓ new 연산자와 Cat() 생성자 사용

```
Cat kitty = new Cat();
```



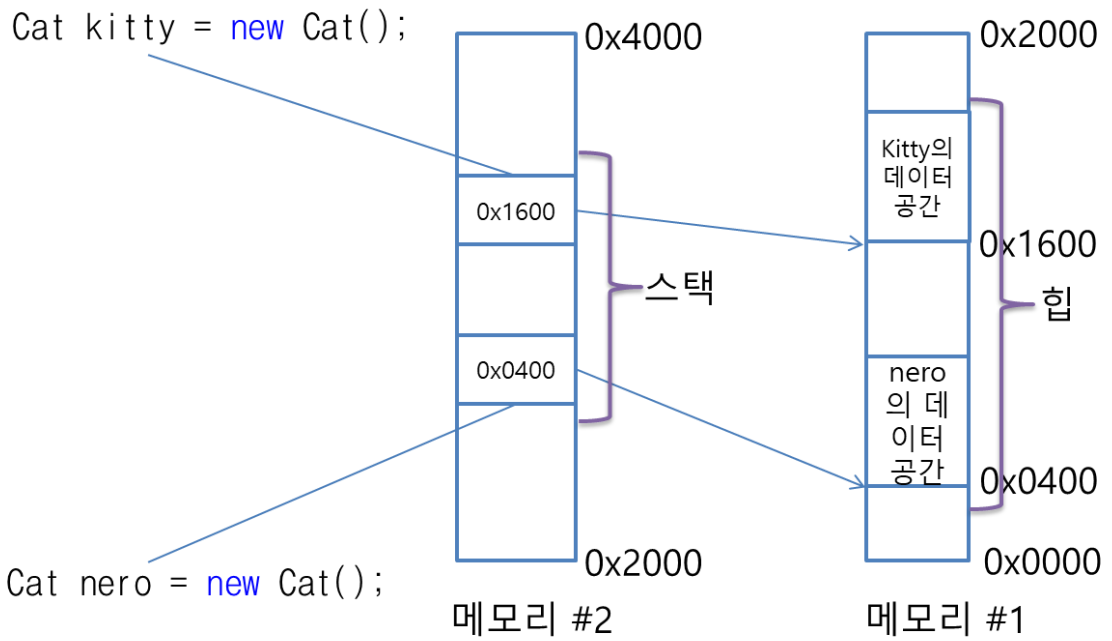
클래스 선언과 객체 생성 예제 코드

```
class Cat
{
    public string Name;      // 필드1
    public string Color;     // 필드2

    public void Meow()       // 메소드
    {
        Console.WriteLine($"{Name} : 야옹");
    }
}
static void Main(string[] args)
{
    Cat kitty = new Cat();
    kitty.Color = "하얀색";
    kitty.Name = "키티";
    kitty.Meow();
    Console.WriteLine($"{kitty.Name} : {kitty.Color}");

    Cat nero = new Cat();
    nero.Color = "검은색";
    nero.Name = "네로";
    nero.Meow();
    Console.WriteLine($"{nero.Name} : {nero.Color}");
}
```

출력 결과:
키티 : 야옹
키티 : 하얀색
네로 : 야옹
네로 : 검은색



클래스 (class)

- 객체지향 프로그래밍
- 클래스 선언과 객체 생성
- 생성자와 종료자
- 메소드

SEJONG UNIVERSITY

생성자

- 객체가 생성 될 때 생성자를 호출
- 생성자 메소드는 클래스와 이름이 같고 반환 형식이 없음
  - ✓ 매개변수를 입력 받아 객체를 생성하는 시점에 필드 초기화 가능
  - ✓ 명시적 구현 없이도 컴파일러에 의해 자동 생성
  - ✓ 개발자가 생성자를 직접 정의한 경우, C# 컴파일러는 생성자를 자동 생성하지 않음
- 오버로딩(Overloading) 사용 가능
  - ✓ 생성자 명(name)은 같지만 입력 받는 매개변수가 다른 다수의 생성자 정의 가능

사용 형식

```
class 클래스이름
{
    한정자 클래스 이름( 매개변수_목록 )
    {
        //
    }

    // 필드
    // 메소드
}
```

생성자 사용 예제

```
class Cat
{
    public Cat()
    { // 매개변수 없는 생성자
        Name = "";
        Color = "";
    }

    public Cat(string _Name, string _Color)
    { // 두 매개변수를 가지는 생성자
        Name = _Name;
        Color = _Color;
    }

    public string Name;      // 필드1
    public string Color;     // 필드2

    // ...
}
```

- Cat 클래스 객체 Kitty와 nabi 생성
  - ✓ Kitty: 매개변수 없는 생성자 이용
  - ✓ nabi: 두 매개변수를 가지는 생성자 이용

```
static void Main(string[] args)
{
    // 첫 번째 생성자 이용
    Cat kitty = new Cat();
    kitty.Color = "하얀색";
    kitty.Name = "키티";

    // 두 번째 생성자 이용 (두 개의 매개변수)
    Cat nabi = new Cat("나비", "갈색");
}
```

종료자

- 객체가 종료 될 때 종료자를 호출
  - ✓ CLR의 가비지 컬렉터가 알아서 객체를 소멸
  - ✓ 종료자는 개발자가 사용하지 않는 것을 권함
- 한정자 없이 클래스 이름 앞에 "~" 표기
- 오버로딩 지원하지 않고 직접 호출 불가능

사용 형식

```
class 클래스이름
{
    ~클래스이름 ( )      // 종료자
    {
        //
    }

    // 필드
    // 메소드
}
```

생성자/종료자 예제 코드

```
출력 결과:
키티 : 야옹
키티 : 하얀색
나비 : 야옹
나비 : 갈색
나비 : 잘가
키티 : 잘가
```

```
class Cat
{
    public Cat()
    { // 매개변수 없는 생성자
        Name = "";
        Color = "";
    }

    public Cat(string _Name, string _Color)
    { // 두 매개변수를 가지는 생성자
        Name = _Name;
        Color = _Color;
    }

    public string Name;     // 필드1
    public string Color;    // 필드2

    public void Meow()      // 메소드
    {
        Console.WriteLine($"{Name} : 야옹");
    }
    ~Cat()      // 종료자
    {
        Console.WriteLine($"{Name} : 잘가");
    }
}
```

```
class MainApp
{
    public void CreatCatObjects()
    {
        Cat kitty = new Cat("키티", "하얀색");
        kitty.Meow();
        Console.WriteLine($"{kitty.Name} : {kitty.Color}");


        Cat nabi = new Cat("나비", "갈색");
        nabi.Meow();
        Console.WriteLine($"{nabi.Name} : {nabi.Color}");
    }

    static void Main(string[] args)
    {
        MainApp MainProgram = new MainApp();
        MainProgram.CreatCatObjects();  // Cat 객체생성 메서드
        GC.Collect();                   // 가비지 컬렉터 수행
        GC.WaitForPendingFinalizers(); // 종료자 큐 처리동안 wait
    }
}
```

클래스 (class)

- 객체지향 프로그래밍
- 클래스 선언과 객체 생성
- 생성자와 종료자
- 메소드
  (return 문, 정적필드/메소드, 매개변수)

메소드 (Method)

- C언어의 함수와 같은 개념이고, C#에서는 클래스 안에 존재
- 매개변수와 반환 형식을 가짐
  - ✓ 매개변수: 메소드 안에서 사용되어지는 변수
  - ✓ 반환 형식: 메소드 수행 결과 값의 데이터형, 수행 결과는 메소드 호출자에게 반환
  - ✓ 반환할 수행결과가 없는 메소드 경우 반환형식으로 “void” 이용

사용 형식

```
class 클래스이름
{
    한정자 반환_형식 메소드_이름( 매개변수_목록 )
    {
        // 실행하고자 하는 코드 1
        // 실행하고자 하는 코드 1
        // …
        // 실행하고자 하는 코드 1

        return 매소드_결과;
    }
}
```

메소드 사용 예제

- 메소드 호출 시 일어나는 프로그램 흐름 (예제코드 참고)
  - ✓ (1) Calculator 객체의 Plus() 메소드 호출, 3과 4에 해당하는 인수를 넘김
  - ✓ (2) 프로그램 흐름이 Plus() 메소드로 이동 후, 메소드 안에 코드를 차례로 수행
  - ✓ (3) 메소드 블록 끝 도달 또는 return 문을 만났을 경우 메소드 종결
  - ✓ (4) Plus() 메소드 호출한 곳으로 흐름이 되돌아 오고 이후 코드를 계속 실행

```
class MainApp
{
    static void Main(string[] args)
    {
        Calculator Calc = new Calculator();

        int x = Calc.Plus(3, 4);

        Console.WriteLine($"3 + 4 = {x}");
    }
}
```

(1)

(3)

(4)

메소드를 포함한 클래스

```
class Calculator
{
    public int Plus (int a, int b)
    {
        int result = a + b;
        return result;
    }
}
```

(2)

메소드 예제 코드

출력 결과:

```
3 + 4 = 7
5 - 2 = 3
```

메소드를 포함한 클래스

```
class Calculator
{
    public int Plus (int a, int b)
    {
        return a + b;
    }

    public int Minus(int a, int b)
    {
        return a - b;
    }
}
```

```
class MainApp
{
    static void Main(string[] args)
    {
        Calculator Calc = new Calculator();

        int x = Calc.Plus(3, 4);
        Console.WriteLine($"3 + 4 = {x}");

        int y = Calc.Minus(5, 2);
        Console.WriteLine($"5 - 2 = {y}");
    }
}
```

return 문

- 프로그램의 흐름을 호출자에게로 돌려 놓음
  - ✓ 메소드 끝 뿐 아니라 메소드 중간에도 사용하여 메소드 종결 가능
  - ✓ 반환 형식이 void 인 경우에도 return 문 사용 가능
- 재귀 호출
  - ✓ 메소드가 자기 자신을 스스로 호출하는 것
  - ✓ 코드를 간결하게 작성 할 수 있지만 성능엔 악영향이 될 수 있음

```
public int Fibonacci (int n)
{
    if (n < 2)
        return n;
    else
        return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

```
public void PrintProfile(string name, string phone)
{
    if (name == "")
    {
        Console.WriteLine("이름을 입력해주세요.");
        return;
    }
    Console.WriteLine($"Name:{name}, Phone:{phone}");
}
```

return문 예제 코드

메소드를 포함한 클래스

```
class ReturnTest
{
    public int Fibonacci(int n)
    {
        if (n < 2)
            return n;
        else
            return Fibonacci(n - 1) + Fibonacci(n - 2);
    }

    public void PrintProfile(string name, string phone)
    {
        if (name == "")
        {
            Console.WriteLine("이름을 입력해주세요.");
            return;
        }
        Console.WriteLine($"Name:{name}, Phone:{phone}");
    }
}
```

출력 결과:
10번째 수: 55
이름을 입력해주세요.
Name:홍길동, Phone:456-1230

```
class MainApp
{
    static void Main(string[] args)
    {
        ReturnTest RT = new ReturnTest();
        Console.WriteLine($"10번째 수: {RT.Fibonacci(10)}");

        RT.PrintProfile("", "123-4567");
        RT.PrintProfile("홍길동", "456-1230");
    }
}
```

정적 (static) 필드와 메소드

- static 키워드
  - ✓ 메소드나 필드가 클래스의 객체가 아닌 클래스 자체에 소속 되도록 지정
  - ✓ 클래스에 소속된다는 것은 프로그램 전체에 유일한 필드라는 의미
  - ✓ 정적 필드를 프로그램 전체에 걸쳐 공유하는 변수로 사용

객체에 소속된 필드

```
class MyClass
{
    public int a;
    public int b;
}

class MainApp
{
    static void Main(string[] args)
    {
        MyClass obj = new MyClass();
        obj.a = 1;
        obj.b = 2;
    }
}
```

클래스(static)에 소속된 필드

```
class MyClass
{
    public static int a;
    public static int b;
}

class MainApp
{
    static void Main(string[] args)
    {
        MyClass.a = 1; // 객체를 생성하지 않고 클래스의
        MyClass.b = 2; // 이름을 통해 필드에 직접 접근
    }
}
```

정적 필드 예제 코드

```
class Global
{
    public static int Count = 0;
}

class ClassA
{
    public ClassA ()
    {
        Global.Count++;
    }
}

class ClassB
{
    public ClassB()
    {
        Global.Count++;
    }
}
```

출력 결과:

```
Global Count : 0
Global Count : 4
```

```
class MainApp
{
    static void Main(string[] args)
    {
        Console.WriteLine($"Global Count : {Global.Count}");
        new ClassA();
        new ClassA();
        new ClassB();
        new ClassB();
        Console.WriteLine($"Global Count : {Global.Count}");
    }
}
```

정적 메소드

- 정적 필드와 마찬가지로 객체가 아닌 클래스 자체에 소속
  - ✓ 보통 객체 내부의 데이터를 이용해야 할 경우 정적 메소드가 아닌 객체 메소드 사용

객체에 소속된 메소드

```
class MyClass
{
    public void InstanceMethod()
    {
        // ...
    }
}

// ,,,

MyClass obj = new MyClass();
obj.InstanceMethod();
// 객체를 생성해야 호출가능
```

클래스(static)에 소속된 메소드

```
class MyClass
{
    public static void StaticMethod()
    {
        // ...
    }
}

// ,,,

MyClass.StaticMethod();
// 객체를 만들지 않고도 바로 호출가능
```

값에 의한 매개변수 전달

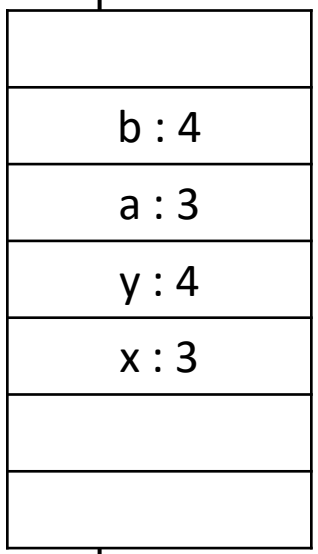
```
class Calculator
{
    public static int Plus(int a, int b)
    {
        Console.WriteLine("Input : {0}, {1}", a, b);

        int result = a + b;
        return result;
    }
}

class MainApp
{
    static void Main(string[] args)
    {
        int x = 3;
        int y = 4;

        int result = Calculator.Plus(x, y);
    }
}
```

int a = x, b = y;



- 메소드에 인수로 넘겨진 값들은 매개변수에 할당됨
  - ✓ 한 변수를 또 다른 변수에 할당하면 데이터 값만 복사
  - ✓ 예제코드에서 변수 a와 x는 똑같은 데이터를 갖지만 별개의 메모리 공간 사용

값에 의한 전달 예제 코드

```
class MainApp
{
    public static void Swap(int a, int b)
    {
        int temp = b;
        b = a;
        a = temp;
    }

    static void Main(string[] args)
    {
        int x = 3;
        int y = 4;

        Console.WriteLine($"x:{x}, y:{y}");
        Swap(x, y);
        Console.WriteLine($"x:{x}, y:{y}");
    }
}
```

- Swap 메소드
  - ✓ 두 정수 입력 값을 매개변수로 전달 받아 변수의 값을 서로 바꿈
  - ✓ 매개변수에는 인수 값이 복사되어 전달
  - ✓ Main 메소드 안의 x와 y에는 영향이 없음

출력 결과:
x:3, y:4
x:3, y:4

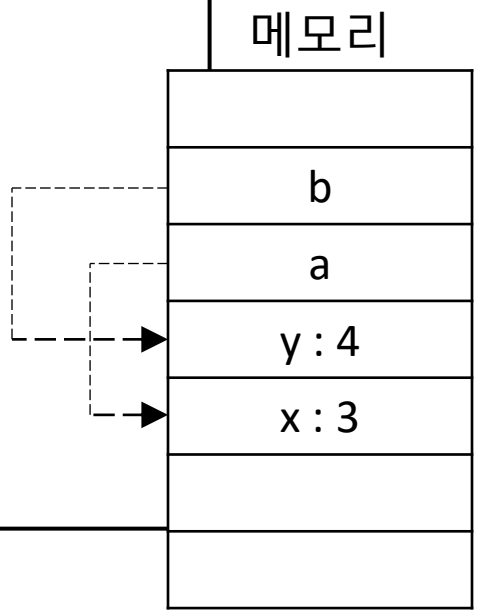
참조에 의한 매개변수 전달

```
public static void Swap(ref int a, ref int b)
{
    int temp = b;
    b = a;
    a = temp;
}
static void Main(string[] args)
{
    int x = 3;
    int y = 4;

    Console.WriteLine($"x:{x}, y:{y}");

    Swap(ref x, ref y);

    Console.WriteLine($"x:{x}, y:{y}");
}
```



- 매개변수가 메소드에 넘겨진 원본 변수를 직접 참조
  - ✓ 메소드 안에서 매개변수를 수정하면 원본 변수에도 수정이 이뤄짐
  - ✓ 메소드를 호출할 때와 메소드를 선언 할 때 매개변수 앞에 ref 키워드를 붙여줌

출력 결과:
x:3, y:4
x:4, y:3

메소드 결과를 참조로 반환

- 메소드의 결과를 참조로 반환하는 참조 반환값
  - ✓ 메소드 호출자로 하여금 반환 받은 결과를 참조로 다룰 수 있도록 함
  - ✓ ref 키워드를 이용해서 메소드 선언,
  - ✓ return 문이 반환하는 변수 앞에도 ref 키워드 명시
  - ✓ 호출자는 결과를 값으로 반환 받을 수 있고, 참조로도 반환 받을 수 있음

참조 반환 값을 사용하는 메소드 작성

```
class SomeClass
{
    int SomeValue = 10;

    public ref int SomeMethod()
    {
        return ref SomeValue;
    }
}
```

값으로 반환 받는 코드

```
SomeClass obj = new SomeClass();
int result = obj.SomeMethod();
```

참조로 반환 받는 코드

```
SomeClass obj = new SomeClass();
ref int result = ref obj.SomeMethod();
```

메소드 결과를 참조로 반환 예제 코드

```
class Product
{
    private int price = 100;

    public ref int GetPrice()
    {
        return ref price;
    }

    public void PrintPrice()
    {
        Console.WriteLine($"Price : {price}");
    }
}
```

```
static void Main(string[] args)
{
    Product carrot = new Product();
    ref int ref_local_price = ref carrot.GetPrice();
    int normal_local_price = carrot.GetPrice();

    carrot.PrintPrice();
    Console.WriteLine($"Ref Local Price : {ref_local_price}");
    Console.WriteLine($"Normal Local Price :{normal_local_price}");

    ref_local_price = 200;

    carrot.PrintPrice();
    Console.WriteLine($"Ref Local Price : {ref_local_price}");
    Console.WriteLine($"Normal Local Price :{normal_local_price}");
}
```

출력 결과:

```
Price : 100
Ref Local Price : 100
Normal Local Price :100
Price : 200
Ref Local Price : 200
Normal Local Price :100
```

Thank you!